BASIC LINUX COMMANDS

ls - list the content of the directory user is currently in

man [command] - information on how to use [command]

info [command] - information on how to use [command]

whatis [command] - short description of the command

ls [directory] - list the content of the [directory]

ls -al [directory] - list the content of the [directory], -a: do not ignore entries starting with ., -l: use a long listing format

pwd - shows the path of the current directory

cd [directory] - move to the [directory]

cd .. - move to the lower directory

cd ../.. - move 2 directories lower

mkdir [directory] - make new directory with name [directory]

rm -r [directory] - delete directory [directory] with all of its content

touch [file] - create a new file with name [file]

cat >[file] - create a new file with name [file] and copy any characters you type at the keyboard to the designated file

cp [file] [directory] - copy [file] to the [directory]

rm [file] - delete the file

more [file] - show the content of the [file]

mv [file] [target] - rename [file] to [target] OR move [file] to [target]

chmod [permissions] [file/directory] - change permissions

• 4 and "r" stand for "read"

• 2 and "w" stand for "write"

• 1 and "x" stand for "execute"

• 0 stands for "no permission"

EXERCISE 1

Using the Linux terminal generate a file structure which looks like this.

Write all commands you use in a separate file using LibreOffice Writer.

Desktop ------- CP ---- Cities

| |

| ---------- Streets

|

--------- Villages

|

---------Towns

In the directory Towns generate a file Piran.txt.

Create a file Kettejeva\_ulica.txt in the directory Desktop and move it to the directory Streets.

Create a file Korte.txt in the directory Desktop and copy it to the directory Villages.

Directory Cities and everything in it can be accessed (read and execute!) to everybody.

Directories Villages and Towns are only accessible (all permissions) to the owner.

(\*SOLUTION\*)

1. Create the initial directories:

mkdir -p ~/Desktop/CP/Cities

mkdir -p ~/Desktop/CP/Streets

mkdir -p ~/Desktop/Villages

mkdir -p ~/Desktop/Towns

2. Generate the file Piran.txt in the Towns directory:

touch ~/Desktop/Towns/Piran.txt

3. Create the file Kettejeva\_ulica.txt in the Desktop directory:

touch ~/Desktop/Kettejeva\_ulica.txt

4. Move Kettejeva\_ulica.txt to the Streets directory:

mv ~/Desktop/Kettejeva\_ulica.txt ~/Desktop/CP/Streets/

5. Create the file Korte.txt in the Desktop directory:

touch ~/Desktop/Korte.txt

6. Copy Korte.txt to the Villages directory:

cp ~/Desktop/Korte.txt ~/Desktop/Villages/

REGULAR EXPRESSIONS

• Regular expression, regex or regexp is a formal language theory, a sequence of characters that define a search pattern. This pattern is usually used by string searching algorithms for „find“ or „find and replace“ operations on strings. (source: Wikipedia)

• Different tools which use regex: Google Code Search (shut down in March 2013), grep, IDE code competition, Kate, ed, find, locate, vi, emacs, .NET, Java SDK, Exalead, etc

Every sign represents itself except [\^$.|?\*+(){}

• \ followed by [\^$.|?\*+(){} represents the following sign

• [str] represents one sign from the set in the brackets: s, t or r

• [^str] represents negation of the set – sign which is not s, t or r

• - (except before or after [) represents a range: [a-zA-Z0-9] all numbers and charactes

• [-] represents a minus

• (str) represents a substring/subexpression which can be recalled later

• | represents a choice (also known as alternation or set union): a|b means a or b; a(b|c)d means abd or acd

• . matches any single character (except space or new line): a.cd can be abcd, aXcd,...

• [.] represents a fullstop

• \* after a sign means zero or more repretitions of the sign: ab\*c can be: ac, abc, abbc,... a(bb)\*c can be: ac, abbc, abbbbc,... [xyz]\* can be ‘‘, x, y, zx, zyx,...

• + after a sign means one or more repetitions of the sign: ab+c can be: abc, abbc,...

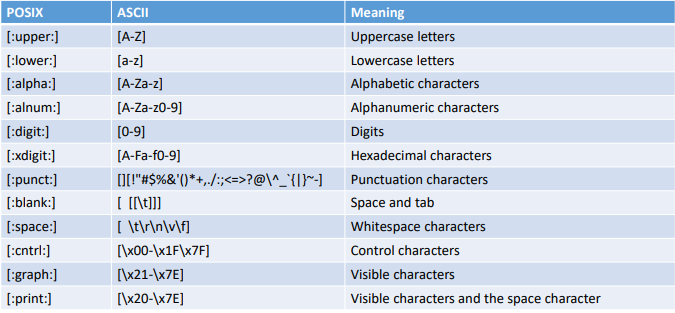
• [xyz]+ can be: x, y, zx, zyx,...

• ? means previous sign (string) can be present or not: ab?c can be ac or abc

{} limiting the number of repetitions of a previous sign (subset) {n} repetition of a previous sign exactly n-times: a{3} means aaa {n,m} repetition of a previous sign at least n-times and at most mtimes {n,} repetition of a previous sign at least n-times

• ^ represents start of a string

• $ represents end of the string



Example 1

(a|b)\*ccc represents strings which can start with any number of repetitions of a letter a and any number of repetitions of a letter b and end with three repetitions of a letter c.

• Sign | separates alternatives: a|b means „a or b“

• Sign \* means zero or more repetitions of the expression before the sign: (a|b)\* means „any number of repetitions of a or b“

• (subset): brackets are used to limit the subset • At the end there are 3 repetitions of a letter c ccc cccccc (wrong!) bbbbaccc ababaaccc cccaababa (wrong!) acccc

Example 2

• (Luis Fonsi)|(luis fonsi) Luis Fonsi luis fonsi

• (L|l)uis (F|f)onsi Luis Fonsi Luis fonsi luis Fonsi luis fonsi

• (a\*)b(a\*)b(a\*)b(a\*) All strings of a-s and b-s where b is repeted exactly three times

Example 3

• 0|((1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)\*) String 0 and all strings of digits which don‘t start with 0.

• 0|([1-9][0-9]\*) String 0 and all strings of digits which don‘t start with 0.

• [A-Z][a-z]\* All strings of letters which start with a capital letter.

• [A-Da-z]\* Strings which contain letters A, B, C and D and small letters, example: aaaBfdCDsdfsdAzz.

EXERCISES 2 - 7

2. On the desktop go to the directory we have created last time

• Create new directory bbb.txt

• Move to the directory bbb.txt

• Use editor emacs and create a file names.txt

• How? Try typing in emacs names.txt

• Check what is stored in a file names.txt

(\*SOLUTION\*)

cd ~/Desktop/CP/Streets

mkdir bbb

cd bbb

emacs names.txt

cat names.txt

3. Open file names.txt with nano editor and add name Vida to the file, save and close it.

• Check what is written to the file names.txt

• Use regular expressions and egrep command to find all starting lines in a file names.txt.

• How does egrep command work? How do we figure that out?

(\*SOLUTION\*)

nano names.txt

cat names.txt

egrep '^pattern' names.txt

man egrep

4. Find all lines in the file names.txt starting with M.

• Find lines starting with M and ending with a with any number of letters in between.

• Find lines ending with a.

• Lines ending with a and have at least three letters before the a.

(\*SOLUTION\*)

egrep '^M' names.txt

egrep '^M.\*a$' names.txt

egrep 'a$' names.txt

egrep '^...a$' names.txt

5. Lines ending with a and have exactly 4 letters.

• Lines which start with either M or L

• Write lola to the end of a file names.txt without using the editor.

• Write LOLA to the end of a file names.txt without an editor.

• Show the content of a file to the screen.

(\*SOLUTION\*)

egrep '^...a$' names.txt

egrep '^[ML]' names.txt

echo "lola" >> names.txt

echo "LOLA" >> names.txt

cat names.txt

6. Print out all lines containing name lola where every letter can be a small or a big letter.

• Lines starting with Marjan and have 0 or more letters after it.

• Lines starting with Marjan and have at least one letter after it.

• Lines starting with Marjan and have one or 0 letters after it.

(\*SOLUTION\*)

egrep -i 'lola' names.txt

egrep '^Marjan.\*' names.txt

egrep '^Marjan.' names.txt

egrep '^Marjan.?' names.txt

7. Print the detailed content of the directory bbb.txt.

• Print out detailed content of the directory bbb.txt where the results finish with .txt

• What does locate command do?

• Use locate command and regular expression to find all files finishing with .txt.

(\*SOLUTION\*)

ls -l bbb.txt

ls -l bbb.txt/\*.txt

locate -r '\.txt$'

MORE LINUX COMMANDS

Search and sort

• find search for files in a directory hierarchy First define a directory from where on we are searching. The search is performed recursively. If searching by a name, use switch –name. If searching by a type, use switch –type. If searching by a time change, use switch –mtime, -ctime, -atime, -mmin, -cmin, -amin (and add time you are interested in) (m – modified, c – changed, a – accessed; time is set in days and min in minutes)

• locate Locate command performs search in a database which is generated once a day using updatedb command. When indexing the data it does not look after permissions but it actually index all the files on the disk. When printing out the results it takes into account the permissions a user has on the data. Downside: it does not work in real-time

• ls Results when ls command is performed can be sorted by:

• Ending: -X (sorts by alphabetical order of endings)

• Without sorting: -U (list entries in directory order – directories first)

• Size: -S (ls –lS)

• Version: -v

• Time: -t (sort by time; in combination with -c and -u)

• Changes of status: -c

• ls -ltc (sort by time a change has been made and print it out),

• ls -lc (sort by name and show the time a change has been made)

• ls -c (sort by time)

• Access time: -u

• ls -ltu (sort by access time and prit it out)

• ls -lu (show access time and sort by name)

• ls –u (sort by access time) Search and sort

sort lines of text files By default it sorts lines by alphabetical order. To sort by „numbers“, use –n usage: sort names.txt Time commands

• date show and set time and date

• cal printout of a calendar by different criteria (years, months,...)

• time put it before any other command to record the time to execute the given command

• crontab automatic scheduler (a list of commands that you want to run on a regular schedul) Archiving • tar (tape archiver) programme for archiving (combining several files and directories in one archive file) • gzip, gunzip compression and un-compression of documents

• There are several more programmes which can compress the data (zip, unzip, rar, unrar,...) User controls

• passwd: change the password (this does not work @FAMNIT computers due to different kind of authentication)

• id, groups: id number of a user and which groups does a user belong to

• finger: infomation about the user

• whoami, who: information on signed-in users

• su, sudo: log-in and execution of commands under a different username

• adduser, addgroup: adding users and groups

• usermod, userdel: editing of user‘s information and deleting of users Processes

• top: display linux processes and resource usage in real time

• uptime: display how long the system has been running

• ps, pstree: display a tree of processes

• kill: stop the execution of a process

• uname: print system information (OS information)

• cat /proc/cpuinfo: information on processor Network

• ifconfig, iwconfig: data and setting of the data on network cards

• route: printing, adding and deleting of data from routing table

• hostname:printing and setting the name of the computer

• /etc/resolv.conf: listed domain servers (each in it‘s own line)

------------------------------------------------------------------------------------------------------------------------------------------

• mount, unmount: connecting and disconnecting a device

• shutdown, (halt, reboot): shutting down the computer

Introduction to Bash scripts

Boolean operations

1. #!/bin/bash

2. # and example

3.

4. if [ -r $1 ] && [ -s $1 ]

5. then

6. echo This file is useful.

7. fi 1. #!/bin/bash

2. # or example

3.

4. if [ $USER == 'bob' ] || [ $USER == 'andy' ]

5. then

6. ls -alh

7. else

8. ls

9. fi Sometimes we only want to do something if multiple conditions are met. Other times we would like to perform the action if one of several condition is met. We can accommodate these with boolean operators.

and - && or - ||

Case statements

1. #!/bin/bash

2. # case example

3.

4. case $1 in

5. start)

6. echo starting

7. ;;

8. stop)

9. echo stoping

10. ;;

11. restart)

12. echo restarting

13. ;;

14. \*)

15. echo don\'t know

16. ;;

17. esac

If we wish to take different paths based upon a variable matching a series of patterns, we could use a series of if and elif statements but that would soon be unclear. To make things cleaner, we can use case statement.

case in ) ;; ) ;; esac

Case statements (2)

1. #!/bin/bash

2. # case example

3.

4. case $1 in

5. start)

6. echo starting

7. ;;

8. stop)

9. echo stoping

10. ;;

11. restart)

12. echo restarting

13. ;;

14. \*)

15. echo don\'t know

16. ;;

17. esac

• Line 4 - This line begins the case mechanism.

• Line 5 - If $1 is equal to 'start' then perform the subsequent actions. the ) signifies the end of the pattern.

• Line 7 - We identify the end of this set of statements with a double semi-colon ( ;; ). Following this is the next case to consider.

• Line 14 - Remember that the test for each case is a pattern. The \* represents any number of any character. It is essentially a catch all if none of the other cases match. It is not necessary but is often used.

• Line 17 - esac is case backwards and indicates we are at the end of the case statement. Any other statements after this will be executed normally.

EXERCISE 8

Write a script, using case statement to perform basic math operation as follows: + addition - subtraction x multiplication / division The name of script must be 'q4' which works as follows: $ ./q4 20 / 3, Also check for sufficient command line arguments

(\*solution\*)

#!/bin/bash

# Check for sufficient command-line arguments

if [ "$#" -lt 3 ]; then

echo "Insufficient command-line arguments."

echo "Usage: ./q4 <operand1> <operator> <operand2>"

exit 1

fi

operand1=$1

operator=$2

operand2=$3

# Perform the math operation based on the operator

case $operator in

+)

result=$((operand1 + operand2))

echo "Result: $operand1 $operator $operand2 = $result"

;;

-)

result=$((operand1 - operand2))

echo "Result: $operand1 $operator $operand2 = $result"

;;

x)

result=$((operand1 \* operand2))

echo "Result: $operand1 $operator $operand2 = $result"

;;

/)

result=$(bc <<< "scale=2; $operand1 / $operand2")

echo "Result: $operand1 $operator $operand2 = $result"

;;

\*)

echo "Invalid operator: $operator"

echo "Supported operators: +, -, x, /"

;;

esac

------------------------------------------------------------------------------------------------------------------------------------------

While loops

1. #!/bin/bash

2. # Basic while loop

3.

4. counter=1

5. while [ $counter -le 10 ]

6. do

7. echo $counter

8. ((counter++))

9. done While an expression/test (the test is placed between square brackets [ ]) is true, keep executing lines of code: while [ ] do done

Line 4 – Initialization of the variable counter with a starting value.

Line 5 – While the test is true (counter is < or = to 10) do the following commands.

Line 7 – Print value of the variable counter.

Line 8 – Increase the value of counter by 1.

Line 9 - We're at the bottom of the loop, go back to line 5 and perform the test again. If the test is true then execute the commands. If the test is false then continue executing any commands following done

Until loops

1. #!/bin/bash

2. # Basic until loop

3.

4. counter=1

5. until [ $counter -gt 10 ]

6. do

7. echo $counter

8. ((counter++))

9. done The until loop is similar to the while loop. The difference is that it will execute the commands within it until the test becomes true. until [ ] do done

Line 5 – Until the test is true (counter is > 10) do the following commands.

Line 7 – Print value of the variable counter.

Line 8 – Increase the value of counter by 1.

EXERCISES 9 - 11

9. Write a script to print numbers as 5, 4, 3, 2, 1 using while loop.

(\*SOLUTION\*)

#!/bin/bash

count=5

while [ $count -gt 0 ]; do

echo $count

((count--))

done

10. Write a script to print a given number in a reverse order (eg. if a number is 123 it must print as 321).

(\*SOLUTION\*)

#!/bin/bash

# Check for command-line argument

if [ "$#" -ne 1 ]; then

echo "Usage: ./reverse\_number.sh <number>"

exit 1

fi

number=$1

reverse=""

# Reverse the number

while [ $number -gt 0 ]; do

remainder=$((number % 10))

reverse="${reverse}${remainder}"

number=$((number / 10))

done

echo "Reversed number: $reverse"

11. Write a script to print a sum of all digits of a given number (eg. if a

number is 123 it's sum of all digit will be 1+2+3 = 6).

(\*SOLUTION\*)

#!/bin/bash

# Check for command-line argument

if [ "$#" -ne 1 ]; then

echo "Usage: ./sum\_of\_digits.sh <number>"

exit 1

fi

number=$1

sum=0

# Calculate the sum of digits

while [ $number -gt 0 ]; do

remainder=$((number % 10))

sum=$((sum + remainder))

number=$((number / 10))

done

echo "Sum of digits: $sum"

------------------------------------------------------------------------------------------------------------------------------------------

What is a Bash script?

• A script tells a computer what it should do or say.

• With a Bash script we are telling the Bash shell what it should do.

• A Bash script is a plain text file which contains a series of commands.

• Anything you can run normally on the command line can be put into a script and it will do exactly the same thing.

• By convention we give the Bash script files an extension .sh.

How can I run a Bash script?

• Before running a script we have to set the execute permission. (for safety reasons this permission is generally not set by default).

• To run the script, type in: ./scriptName.sh

• If you just type a name on the command line, Bash tries to find it in a series of directories stored in a variable called $PATH and doesn't consider sub directories or your current directory.

• We can see the current value of this variable using the command echo.

• If a script is not in one of the directories in your $PATH then you can run it by telling Bash where it should look to find it.

• You can include an absolute or relative path in front of the program or script name.

• The dot ( . ) is a reference to your current directory. Assuming this script is in the home directory you can also run it by using an absolute path

How does a Bash script look like?

1. #!/bin/bash

2. # A sample Bash script

3.

4. echo Hello World!

• Line 1 - Is what's referred to as the shebang.

• #!/bin/bash is the first line of the script. The hash exclamation mark ( #! ) character sequence is referred to as the Shebang. Following it is the path to the interpreter that should be used to run (or interpret) the rest of the lines in the text file. (For Bash scripts it will be the path to Bash.)

• The shebang must be on the first line of the file (line 2 won't do, even if the first line is blank).

• There are no spaces before the # or between the ! and the path to the interpreter.

• Line 2 - This is a comment. Anything after # is not executed. It is for our reference only.

• Line 4 - Is the command echo which will print a message to the screen.

Variables

• A variable is a temporary store for a piece of information. There are two actions we may perform for variables:

• Setting a value for a variable.

• Reading the value for a variable.

• To read the variable we place its name (preceded by a $ sign) anywhere in the script we would like.

• Before Bash interprets every line of our script it first checks to see if any variable names are present.

• For every variable it has identified, it replaces the variable name with its value. Then it runs that line of code and begins the process again on the next line.

• Syntax:

• When referring to or reading a variable we place a $ sign before the variable name.

• When setting a variable we leave out the $ sign.

• Some people write variable names in uppercase so they stand out. It's your preference. Command line arguments

1. #!/bin/bash

2. # A sample Bash script

3.

4. cp $1 $2

5.

6. # Let‘s verify the values

7.

8. echo Details for $2

9. ls –l $2

• To supply arguments to a script, we use the variables $1 to represent the first command line argument, $2 to represent the second command line argument and so on.

• Line 4 - run the command cp with the first command line argument as the source and the second command line argument as the destination.

• Line 8 - run the command echo to print a message.

• Line 9 - After the copy has completed, run the command ls for the destination just to verify it worked. We have included the options l to show us extra information so we can verify it copied correctly. Special variables $0 - The name of the Bash script. $1 - $9 - The first 9 arguments to the Bash script. $# - How many arguments were passed to the Bash script. $@ - All the arguments supplied to the Bash script. $? - The exit status of the most recently run process. $$ - The process ID of the current script. $USER - The username of the user running the script. $HOSTNAME - The hostname of the machine the script is running on. $SECONDS - The number of seconds since the script was started. $RANDOM - Returns a different random number each time is it referred to. $LINENO - Returns the current line number in the Bash script. If you type the command env on the command line you will see a listing of other variables which you may also refer to.

Setting our own variables

1. #!/bin/bash

2. # A simple variable example

3.

4. myvariable=Hello

5.

6. anothervar=Mike

7.

8. echo $myvariable $anothervar

9. echo

10.

11. sampledir=/etc

12.

13. ls $sampledir We may also set our own variables. There are a few ways in which variables may be set but this is the basic form: variable=value

• There is no space on either side of the equals (=) sign. We also leave the $ sign from the beginning of the variable name when setting it.

• Lines 4 and 6 - set the value of the two variables myvariable and anothervar.

• Line 8 - run the command echo to check the variables.

• Line 9 - run the command echo without arguments to get a blank line on the screen.

• Line 11 - set a variable as the path to a particular directory.

• Line 13 - run the command ls substituting the value of the variable sampledir as its first command line argument.

EXERCISES 12/3

12. Perform number calculation in bash script and store result to third variable, lets say a=5, b=8, c=a+b?

(\*SOLUTION\*)

#!/bin/bash

a=5

b=8

# Perform the calculation

c=$((a + b))

echo "Result: c = $c"

13. Perform number calculation in bash script and store result to third variable. Put the values of the numbers as an argument to a script.

(\*SOLUTION\*)

#!/bin/bash

# Check for command-line arguments

if [ "$#" -ne 2 ]; then

echo "Usage: ./number\_calculation.sh <a> <b>"

exit 1

fi

a=$1

b=$2

# Perform the calculation

c=$((a + b))

echo "Result: c = $c"

Quotes

When we want variables to store more complex values, we need to make use of quotes. Under normal circumstances Bash uses a space to determine separate items.

• When we enclose our content in quotes we are indicating to Bash that the contents should be considered as a single item. You may use single quotes ( ' ) or double quotes ( " ).

• Single quotes will treat every character literally.

• Double quotes will allow you to do substitution (that is include variables within the setting of the value).

Command substitution

Command substitution allows us to take the output of a command or program (what would normally be printed to the screen) and save it as the value of a variable. To do this we place it within brackets, preceded by a $ sign.

• Command substitution is simple if the output of the command is a single word or line. If the output goes over several lines then the newlines are simply removed and all the output is in a single line.

• Line 1 - Run the command ls. Normally its output would be over several lines.

• Line 4 - When we save the command to the variable myvar all the newlines are stripped out and the output is all on a single line.

User input

1. #!/bin/bash

2. # Ask the user for their name

3.

4. echo Hello, who am I talking to?

5.

6. read varname

7.

8. echo It\'s nice to meet you $varname

Use a command called read, to ask the user for an input. It takes the input and saves it into a variable. read var1

• Line 4 - Print a message asking the user for input.

• Line 6 - Run the command read and save the users response into the variable varname.

• Line 8 - echo another message to verify the read command worked.

1. #!/bin/bash

2. # Ask the user for login details

3.

4. read -p 'Username: ' uservar

5. read -sp 'Password: ' passvar

6. echo

7. echo Thank you $uservar we now have your login details

To alter the behaviour of read there are a variety of command line options. Two commonly used options are: -p to specify a prompt and -s makes the input silent. This can make it easy to ask for a username and password combination (see example)

1. #!/bin/bash

2. # Demonstrate how read works

3.

4. echo What cars do you like?

5. read car1 car2 car3

6.

7. echo Your first car was: $car1

8. echo Your second car was: $car2

9. echo Your third car was: $car3

We can supply several variable names to read. It will take the input and split it on whitespaces.

• The first item will be assigned to the first variable name, the second item to the second variable name and so on.

• If there are more items than variable names, the remaining items will all be added to the last variable name.

• If there are less items than variable names, the remaining variable names will be set to blank or null.

Reading from STDIN

1. #!/bin/bash

2. # A basic summary of my sales report

3.

4. echo Here is a summary of the sales data:

5. echo ==============================

6. echo

7.

8. cat /dev/stdin | cut -d' ' -f 2,3 | sort It's common in Linux to pipe a series of simple, single purpose commands together to create a larger solution tailored to our exact needs. We can use this mechanism with our scripts as well.

• Bash accomodates piping and redirection using special files.

• Each process gets it's own set of files (one for STDIN, STDOUT and STDERR) and they are linked when piping or redirection is invoked :

• STDIN - /dev/stdin or /proc/self/fd/0

• STDOUT - /dev/stdout or /proc/self/fd/1

• STDERR - /dev/stderr or /proc/self/fd/2

• Line 8 - cat the file representing STDIN, cut setting the delimiter to a space, fields 2 and 3 then sort the output.

1. #!/bin/bash

2. # A basic summary of my sales report

3.

4. echo Here is a summary of the sales data:

5. echo ==============================

6. echo

7.

8. cat /dev/stdin | cut -d' ' -f 2,3 | sort

------------------------------------------------------------------------------------------------------------------------------------------

Arithmetic: function let

1. #!/bin/bash

2. # Basic arithmetic using let

3.

4. let a=5+4

5. echo $a # 9

6.

7. let "a = 5 + 4"

8. echo $a # 9

9.

10. let a++

11. echo $a # 10

12.

13. let "a = 4 \* 5"

14. echo $a # 20

15.

16. let "a = $1 + 30"

17. echo $a # 30 + first command line argument let is a built-in function of Bash that allows us to do simple arithmetic. let As you can see in the example, it can take a variety of formats. The first part is a variable which the result is saved into.

• Line 4 - This is the basic format. If we don't put quotes around the expression, it must be written without spaces.

• Line 7 – Quotes allow us to use space in the expression to make it more readable.

• Line 10 – A shorthand for increment the value of the variable a by 1. It is the same as writing "a = a + 1".

• Line 16 - We may also include other variables in the expression.

Arithmetic: function expr

1. #!/bin/bash

2. # Basic arithmetic using expr

3.

4. expr 5 + 4 # 9

5.

6. expr "5 + 4" # 5 + 4

7.

8. expr 5+4 # 5+4

9.

10. Expr 11 % 2 # 1

11.

12. expr 5 \\* $1 # result of (5 \* first arg.)

13.

14. a=$( expr 10 - 3 )

15. echo $a # 7 expr is similar to let except instead of saving the result to a variable it prints out the answer.

• You don't need to enclose the expression in quotes.

• Use spaces between the items of the expression.

• It is common to use expr within command substitution to save the output to a variable. expr item1 operator item2

• Line 4 - The basic format. Note that there must be spaces between the items and no quotes.

• Line 6 - If we put quotes around the expression, the expression will be printed.

• Line 8 - If we do not put spaces between the items of the expression, the expression will be printed.

• Line 10 - Here we demonstrate the operator modulus.

• Line 12 - Some characters have a special meaning to Bash so we must escape them (put a backslash in front of) to remove their special meaning.

• Line 14 - This time we're using expr within command substitution in order to save the result to the variable a.

Arithmetic: double parentheses

1. #!/bin/bash

2. # Basic arithmetic using double parentheses

3.

4. a=$(( 4 + 5 ))

5. echo $a # 9

6.

7. a=$((3+5))

8. echo $a # 8

9.

10. b=$(( a + 3 ))

11. echo $b # 11

12.

13. b=$(( $a + 4 ))

14. echo $b # 12

15.

16. (( b++ ))

17. echo $b # 13

18.

19. (( b += 3 ))

20. echo $b # 16

21.

22. a=$(( 4 \* 5 ))

23. echo $a # 20 To save the output of the basic arithmetic, we can use double brackets: $(( expression ))

• Line 4 - This is the basic format. As you can see we may space it out nicely for readability without the need for quotes.

• Line 7 - It works just the same if we take spacing out.

• Line 10 – We can include variables without preceding $ sign.

• Line 13 - Variables can be included with the $ sign if you prefer.

• Line 16 - Here the value of the variable b is incremented by 1 When we do this we don't need the $ sign before the brackets.

• Line 19 - A slightly different form of the previous example. Here the value of the variable b is incremented by 3. \

• Line 22 - Unlike other methods, when we do multiplication we don't need to escape the \* sign. Length of a variable

1. #!/bin/bash

2. # Show the length of a variable.

3.

4. a='Hello World'

5. echo ${#a} # 11

6.

7. b=4953

8. echo ${#b} # 4

To find out the length of a variable (how many characters) we can use: ${#variable}

IF statements

1. #!/bin/bash

2. # Basic if statement

3.

4. if [ $1 -gt 100 ]

5. then

6. echo Hey that\'s a large number.

7. pwd

8. fi

9.

10. dat

A basic if statement says, if a particular test is true, then perform a given set of actions. If it is not true then don't perform those actions. if [ ] then fi Anything between then and fi (if backwards) will be executed only if the test (between squared brackets) is true.

• Line 4 - If the first cmd. line argument is greater than 100.

• Line 6 and 7 - Will only run if the test on line 4 returns true. You can have as many commands here as you like.

• Line 6 - The backslash ( \ ) in front of the single quote ( ' ) is needed as the single quote has a special meaning for bash and we don't want that special meaning.

• Line 8 - fi signals the end of the if statement. All commands after this will be run as normal.

• Line 10 - This command will be run regardless of the outcome of the if statement.

Test command The square brackets ( [ ] ) in the if statement are actually a reference to the command test. This means that all of the operators that test allows may be used here as well. Some of the more common operators are listed below.

NOTE:

• = is slightly different to -eq. [ 001 = 1 ] will return false as = does a string comparison (ie. character for character the same) whereas -eq does a numerical comparison meaning [ 001 - eq 1 ] will return true.

• When we refer to FILE we are actually meaning a path. Remember that a path may be absolute or relative and may refer to a file or a directory.

• Because [ ] is just a reference to the command test we may experiment and trouble shoot with test on the command line to make sure our understanding of its behaviour is correct.

Line 1 - Perform a string based comparison. Test doesn't print the result so instead we check it's exit status which is what we will do on the next line.

• Line 2 - The variable $? holds the exit status of the previously run command (in this case test). 0 means TRUE (or success). 1 = FALSE (or failure).

• Line 4 - This time we are performing a numerical comparison.

• Line 7 - Create a new blank file myfile (assuming that myfile doesn't already exist).

• Line 8 - Is the size of myfile greater than zero?

• Line 11 - Redirect some content into myfile so it's size is greater than zero.

• Line 12 - Test the size of myfile again. This time it is TRUE.

Nested if statements

1. #!/bin/bash

2. # Nested if statements

3.

4. if [ $1 -gt 100 ]

5. then

6. echo Hey that\'s a large number.

7.

8. if (( $1 % 2 == 0 ))

9. then

10. echo And is also an even number.

11. fi

12. fi

You may have as many if statements as necessary inside your script. It is also possible to have an if statement inside of another if statement.

• Line 4 - Perform the following, only if the first command line argument is greater than 100.

• Line 8 - This is a light variation on the if statement. If we would like to check an expression then we may use the double brackets just like we did for variables.

• Line 10 - Only gets run if both if statements are true.

If - else statement

1. #!/bin/bash

2. # else example

3. # read from a file if it is supplied as

4. # a command line argument, else read

5. # from STDIN.

6.

7. if [ $# -eq 1 ]

8. then

9. nl $1

10. else

11. nl /dev/stdin

12. fi

Sometimes we want to perform a certain set of actions if a statement is true, and another set of actions if it is false. We can accommodate this with the else mechanism. if [ ] then else fi

EXERCISES 14/5

14. Write a bash script that will add two numbers, which are supplied as command line argument, and if this two numbers are not given show error and its usage.

(\*SOLUTION\*)

#!/bin/bash

# Check for command-line arguments

if [ "$#" -ne 2 ]; then

echo "Error: Two numbers are required."

echo "Usage: ./add\_numbers.sh <number1> <number2>"

exit 1

fi

# Retrieve the command-line arguments

number1=$1

number2=$2

# Perform the addition

sum=$((number1 + number2))

echo "Sum: $number1 + $number2 = $sum"

15. Write script to determine whether given file exist or not, file name is supplied as command line argument, also check for sufficient number of command line argument

(\*SOLUTION\*)

#!/bin/bash

# Check for command-line arguments

if [ "$#" -ne 1 ]; then

echo "Error: File name is required."

echo "Usage: ./check\_file.sh <filename>"

exit 1

fi

# Retrieve the file name from command-line argument

filename=$1

# Check if the file exists

if [ -e "$filename" ]; then

echo "File '$filename' exists."

else

echo "File '$filename' does not exist."

Fi

If – elif – else statement

Example: if you are 18 or over you may go to the party. If you aren't but you have a letter from your parents you may go but must be back before midnight. Otherwise you cannot go.

1. #!/bin/bash

2. # elif statements

3.

4. if [ $1 -ge 18 ]

5. then

6. echo You may go to the party.

7. elif [ $2 == 'yes' ]

8. then

9. echo You may go to the party but be back before midnight.

10. else

11. echo You may not go to the party.

12. fi Sometimes we may have a series of conditions that may lead to different paths. if [ ] then elif [ ] then else fi You can have as many elif branches as you like. The final else is also optional.

EXERCISE 16

16. Write Script to find out biggest number from given three numbers. Numbers are supplies as command line argument. Print error if sufficient arguments are not supplied.

(\*SOLUTON\*)

#!/bin/bash

# Check for command-line arguments

if [ "$#" -ne 3 ]; then

echo "Error: Three numbers are required."

echo "Usage: ./find\_biggest.sh <number1> <number2> <number3>"

exit 1

fi

# Retrieve the command-line arguments

number1=$1

number2=$2

number3=$3

# Find the biggest number

biggest=$number1

if [ $number2 -gt $biggest ]; then

biggest=$number2

fi

if [ $number3 -gt $biggest ]; then

biggest=$number3

fi

echo "The biggest number is: $biggest"

------------------------------------------------------------------------------------------------------------------------------------------

For loops

1. #!/bin/bash

2. # Basic for loop

3.

4. names='Alen Marko Aleks'

5.

6. for name in $names

7. do

8. echo $name

9. done

For each of the items in a given list, perform the given set of commands. It has the following syntax: for var in do done The for loop will take each item in the list (in order, one after the other), assign that item as the value of the variable var, execute the commands between do and done then go back to the top, grab the next item in the list and repeat over. The list is defined as a series of strings, separated by spaces.

Line 4 - Create a simple list which is a series of names.

Line 6 - For each of the items in the list $names assign the item to the variable $name and do the following commands.

Line 8 - echo the name to the screen.

For loops - ranges

1. #!/bin/bash

2. # Basic range in for loop

3.

4. for value in {1..5}

5. do

6. echo $value

7. Done

1. #!/bin/bash

2. # Basic range with steps for loop

3.

4. for value in {10..0..2}

5. do

6. echo $value

7. done

We can process a series of numbers. To specify a range use curly braces {No1..No2} in a for loop and specify any number you like for the starting value and the ending value. There are no spaces between the curly braces! The first value may also be larger than the second in which case it will count down. It is also possible to specify a value to increase or decrease by each time. You do this by adding another two dots ( .. ) and the value to step by.

Break & continue

break

The break statement tells Bash to leave the loop straight away. It may be that there is a normal situation that should cause the loop to end but there are also exceptional situations in which it should end as well. For instance, maybe we are copying files but if the free disk space get's below a certain level we should stop copying. continue The continue statement tells Bash to stop running through this iteration of the loop and begin the next iteration. Sometimes there are circumstances that stop us from going any further. For instance, maybe we are using the loop to process a series of files but if we happen upon a file which we don't have the read permission for we should not try to process it. Sometimes we may need to intervene and slightly alter the running of a loop. We can do this using break and continue statements.

Select

1. #!/bin/bash

2. # A symple menu system with select

3.

4. names='Alen Marko Aleks Quit'

5.

6. PS3='Select person: '

7.

8. select name in $names

9. do

10. if [ $name == 'Quit' ]

11. then

12. break

13. fi

14. echo Hello $name

15. done

16.

17. echo Bye The select mechanism allows you to create a simple menu system. It has the following format: select var in do done It takes all the items in list and present them on the screen with a number before each item. A prompt will be printed after this allowing the user to select a number. When they select a number and hit enter the corresponding item will be assigned to the variable var and the commands between do and done are run. Once finished a prompt will be displayed again so the user may select another option.

Line 4 - Set up a variable with the list of names and a last option which we may select to quit.

Line 6 - Change the value of the system variable PS3 with something more descriptive. (By default it is #?)

Lines 10 - 13 - If the last option, 'Quit', is selected break out of the select loop.

EXERCISES

Write a script using for loop to print the following patterns on the screen:

a) 1

22

333

4444

55555

-----------------------

b) 1

12

123

1234

12345

------------------------------

c) \*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

----------------------------------

d) \*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

------------------------------

e) 1

22

333

4444

55555

666666

7777777

88888888

-----------------------------

f) \*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*

(\*SOLUTIONS\*)

a)#!/bin/bash

for ((i=1; i<=5; i++))

do

for ((j=1; j<=i; j++))

do

echo -n "$i"

done

echo

Done

--------------------------------

b) #!/bin/bash

for ((i=1; i<=5; i++))

do

for ((j=1; j<=i; j++))

do

echo -n "$j"

done

echo

Done

-----------------------------------

c)#!/bin/bash

for ((i=1; i<=5; i++))

do

for ((j=1; j<=i; j++))

do

echo -n "\*"

done

echo

Done

----------------------------------------

d)#!/bin/bash

for ((i=5; i>=1; i--))

do

for ((j=1; j<=i; j++))

do

echo -n "\*"

done

echo

Done

----------------------------------------------

e)#!/bin/bash

for ((i=1; i<=8; i++))

do

for ((j=1; j<=i; j++))

do

echo -n "$i"

done

echo

Done

------------------------------------------------------

f)#!/bin/bash

for ((i=1; i<=8; i++))

do

for ((j=1; j<=i; j++))

do

echo -n "\*"

done

echo

done